CMU 18-447: Introduction to Computer Architecture

Handout 8/Project 1: Single-Cycle MIPS

Due: week of 2/23

(150 points, in groups of 2 or 3)

Objectives. In this project, you will become familiar with the MIPS instruction set by implementing a single-cycle core in Verilog. You have two weeks for this project. This is not a lab you want to wait until the night before to tackle. We will award 5% bonus for checking off on Tuesday 2/24.

Project Overview. You will be given the design skeleton of a single-cycle MIPS processor that is capable of performing the ADDI and the SYSCALL instructions. Complete the design of the single-cycle implementation using **synthesizable Verilog** (i.e., your design must pass through Synopsys correctly).

If you have any questions about the correct behavior of the processor, refer to the behavior of **`spim447`** or **`xspim447`** on the ECE Linux workstations (/afs/ece/class/ece447/bin). The SPIM simulators are the "golden" MIPS models for this course. Except for a few changes that we make to keep the project simpler, the architectural-level behavior of your processor designs should agree with SPIM. Good performance is not a consideration for this single-cycle processor. Correctness and synthesizability count for most of the credit in this project. Read this handout carefully to avoid any unnecessary headaches.

Review Lecture 8 thoroughly before you do any thing. Next, make sure you really understand the entire in's and out's of the skeleton code. We highly suggest drawing a diagram of the skeleton datapath before writing any Verilog. A good milestone for the first week of the project is to complete the core diagram and discuss it with a TA. You will be using your core for the remainder of the semester, so you'll want to avoid any quick-and-dirty hacks that can come back to haunt you. Now is a good time to form new groups, but we highly recommend that you stay in the same group for the remaining projects (although you will be allowed to regroup in between each project).

Supported Instructions. Your single-cycle processor will execute one instruction per cycle (as presented in lecture). The required instructions are given in Table 1. While the table appears to have many instructions, there are actually only a few unique instruction behaviors with a number of minor variations. You should tackle the instructions in groups: R-type ALU, I-type ALU, LD, SW, Jump, Branch, etc. The Blackboard has a copy of the *MIPS R4000 User Manual*, which contains the official definition for the instruction set. By default, SPIM simulates a "virtual machine" form of MIPS that lacks delay slots. In particular for this project, **branch offsets are based on the current PC**, not PC+4. You should **not** try to use SPIM's "bare" mode in Project 1. You should **not** implement load or branch delay slots in this project. (This will change in later projects!) Our test cases for Project 1 will always have NOPs and other benign instructions immediately after loads and branches. You are not responsible for instructions that do not appear in Table 1. If an instruction produces an exception, the processor should simply halt (for now).

Table 1: Instructions required for the single-cycle MIPS processor

J	JAL	BEQ	BNE	BLEZ	BGTZ
ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI
XORI	LUI	LB	LH	LW	LBU
LHU	SB	SH	SW	BLTZ	BGEZ
BLTZAL	BGEZAL	SLL	SRL	SRA	SLLV
SRLV	SRAV	JR	JALR	SYSCALL	ADD
ADDU	SUB	SUBU	AND	OR	XOR
NOR	SLT	SLTU			

Project Handout. The files for this project are available in: **`/afs/ece/class/ece447/handout/project1.`** The Verilog code that you are starting with is significantly more complicated than in the previous labs. A summary of the supplied modules is given next.

- **`mips447_struct.v`**: This file contains the top-level skeleton of the MIPS processor core. The initial design is sufficiently complete that you can use the **`addiu.s`** file to test the already implemented **`addiu`** and **`syscall`** instructions as soon as you correctly plug in the register file. To complete the project for the MIPS instructions in Table 1, you will need to make modifications including adding constants, decode logic, registers, muxes, control path and datapath.
- **`reg_file.v`**: a 3-ported register file. This is similar to your design in Lab1 with the addition of an input signal named "**`halted`**". When "halted" is asserted, a non-synthesizable behavior will print the contents of the register file to the screen and to a file "**`regdump.txt`**". The automatic grading script will be looking for correct register file contents at the end of each run. You will need to figure out how to plug this 3-ported register file into **`mips447_struct.v`** to get started. **Unlike your Lab1 register file, the supplied register file supports combinational reads. Further, the supplied register file does not support data forwarding between simultaneous read and write. In project 2, 3 and 4, you will be given another register file description with synchronous read, matching the specification in Lab 1.**
- **`mips_mem_sync.v`**: a dual-ported multi segmented memory module. Before running any simulation, you need to assemble the **`.s`** assembly files using **`spim447`** to generate the memory image files expected by this module. (We discuss the assembly procedure in more detail later.) You should not have to change this file.
- **`mips_defines.v`**: Verilog "defines" for various opcodes, instruction mnemonics, and other useful constants. This file is more complete than necessary for this project. You only need to support the instructions in Table 1.
- **`testjig_core.v`**: The file contains the top-level testjig module that combines the processor core and memory and drives them with a clock and reset signal.

To complete the project, you will make modifications to these files. This can include customizing the interfaces of modules that have been provided. For example you will very likely want to expand the interface we defined for the instruction decoding logic. There are very little limitations to what you can do except:

1. The **`mips_core`** module must remain synthesizable (after removing the non-synthesizable test and debug fixtures).
2. The register file must dump out its contents in the correct format at the end of each simulation.
3. You must do your own work. (See Handout #1 on Academic Honesty.)

Test Cases. Once you plug in the register file correctly, you should be able to immediately test the design in ModelSim using the **`addiu.s`** test case provided. You can check the correctness by comparing the resulting **`regdump.txt`** file and the reference **`addiu_regdump.txt`**. For the demo, you will be expected to run a number of supplied programs (some revealed before the demo, some not). The available testcases are in **`/afs/ece/class/ece447/handout/project1/tests/`** along with their expected final register file contents. In addition, you will want to build a suite of test programs to verify the operations of new capabilities in your implementation as you add them. We have no problem with you sharing these programs with other students. Feel free to post your test programs on the blackboard.

Usually the command to assemble memory images from a MIPS assembly **`.s`** file using SPIM is:

```
spim447 -vasm filename.s mem
```

Due to the limited intruction supportted in Project 1, you need to assemble with the "–notrap" option, i.e.,

```
spim447 –notrap -vasm filename.s mem
```

This option omits the standard exception handler and start-up code, which you cannot yet execute. After you have all instructions implemented, you would not have to use this flag anymore.

The SPIM assembler creates five memory image files: user data, user text, kernel data, kernel text, and stack space. The memory module in **`mips_mem_sync.v`** knows how to deal with these files appropriately. You should take a look at these files, but don't worry too much if you don't understand how they are "magically" integrated into your simulated memory.

Diagram. As in the previous labs, you will have to turn in a computer-drawn diagram of your single cycle processor for the hand-in. Your diagram should be at the same level of detail as you see in the textbook and lecture notes. All major structures (i.e., registers, muxes, incrementers, etc.) should be drawn, as well as boxes for the various control logic blocks. You should label all wires with their names and widths (this includes control path wires). For your sanity, we suggest using different colors (or line styles) to differentiate control and data path connections. We expect that you will base later diagrams off this one, so putting in an extra effort to keep this diagram neat will pay off. Don't be afraid to leave plenty of white space and span multiple sheets of paper!

Handin. You should electronically hand in all of your Verilog files through the course AFS space. Hand in a paper copy of your diagram at your Lab demo period. During the demo, we will ask you questions about your single-cycle design and test it with a number of input programs. All members of the group must attend the demo. During the demo, you need to show that your Verilog code is synthesizable by invoking the synthesis tool. There are two unsynthesizable modules (**`exception_unit`** and **`syscall_unit`**) that should be omitted from synthesis.

Miscellaneous Caveats

• The memory module, in file **`mips_mem.v`** provides four write enables per port. Each write enable corresponds to a byte in the memory. If you want to write a single byte in the memory, you must decode the address to enable only that byte during the write operation.
• Memory accesses should be aligned to the unit of memory that you are accessing: if you are accessing bytes, any address is okay, however if you are accessing 4-byte words, only addresses where the lower two bits are zero are legal.
• There are two modules in **`mips447_struct.v`** file that are not synthesizable. They are the **`syscall_unit`** and the **`exception_unit`**. You should not have to modify these units.
• Be aware that the multiply, divide, and mod operators in Verilog (*, /, and %) are always unsigned. If you are performing a signed operation, you need to add some code. You can still use these operators, just take care. An easiest way to implement the **`mips_ALU`** module is with a behavioral case statement.

Grading

- **Single-cycle MIPS**
  - Proper functionality and synthesis 90 points
  - Q&A during check-off 30 points
- **Block Diagram**
  - Complete block diagram 30 points
- **Total** **150 points**

Late Project Policy. Late labs and projects will lose 10% for each day following your assigned due date and time. The clock stops when all lab materials have been turned in (including Verilog code, diagrams, answers to questions, etc.) and all demos have been completed.